## REMARKS

### Pending Claims

Claims 1, 5, 6, and 8-24 are still pending.

### Claim Rejections Under 35 U.S.C. 103(a)

In this final Office action, the Examiner has rejected all of the pending claims on the same grounds as before, that is, as being unpatentable under 35 U.S.C. 103(a) over Yates, et al. (6,549,959 – "*Yates*"), further in view of statements on pages 1-3 of the applicant's specification.

Delayed exception handling in the invention vs. context switching

The main point of difference between the applicant's invention as defined in the independent claims now present and *Yates* is related to the Examiner's contention that *Yates* teaches:

> delaying handling (col 35, lines 24-59, in context switching technique, when interrupt occurs, the operating system saves the status of the interrupted process, and routes control to the appropriate interrupt handlers) of each asynchronous sensed exception until (col 40, lines 24-40), and no later than, completion of execution of the target instruction sequence corresponding to the current source instruction when the asynchronous exception is sensed (col 40, lines 40-65)

Similarly, as support for stating "applicant's arguments filed on 10/24/03 have been fully considered but they are not persuasive," the Examiner wrote:

> In response to applicants argument that Yates fails to disclose "delaying handling of each asynchronous sensed exception until, and no later than, completion of execution of the target instruction sequence corresponding to the current source instruction when the asynchronous exception is sensed", the examiner respectfully disagrees. The Yates prior art reference teaches the delaying handling by providing context switching. In context switching technique, when interrupt occurs, the operating system saves the status of the interrupted process, and routes control to the appropriate interrupt handlers. Further in context switching the state of an interrupt must be saved and restored (col 40, lines 15-67 and col 41, lines 1-9).

In essence, the Examiner is asserting that context-switching is a form of and is equivalent to delayed handling of asynchronous exceptions as taught in the present invention. This is not so. *Yates* does not delay handling of asynchronous exceptions that occur during execution of a binary-translated instruction sequence with the same roll-*forward* mechanism the invention provides.

The key situation addressed by the novel feature of the invention as defined in the independent claims has two aspects: 1) execution is taking place in binary translation; and 2) an asynchronous exception occurs. The way in which *Yates* handles this situation is described in col. 41, lines 9-25 (emphasis added):

> If an interrupt is directed to something within virtual X86 310, while TAXi code (a **translated** native version of a "hot spot" within an X86 program ...) was running, then the interrupt is handled by **case 353**. Execution is **rolled back** to an X86 instruction boundary. At an X86 instruction boundary, all Tapestry extended context external to the X86 310 is dead, and a relatively simple correspondence between semantically-equivalent Tapestry and X86 machine states can be established. Tapestry execution may be abandoned -- **after the interrupt is delivered**, execution may resume in converter 136. Then, if the interrupt was an asynchronous external interrupt, TAXi will deliver the appropriate X86 interrupt to the virtual X86 **supplying the reconstructed X86 machine state**, and the interrupt will be handled by X86 operating system 306 in the conventional manner.

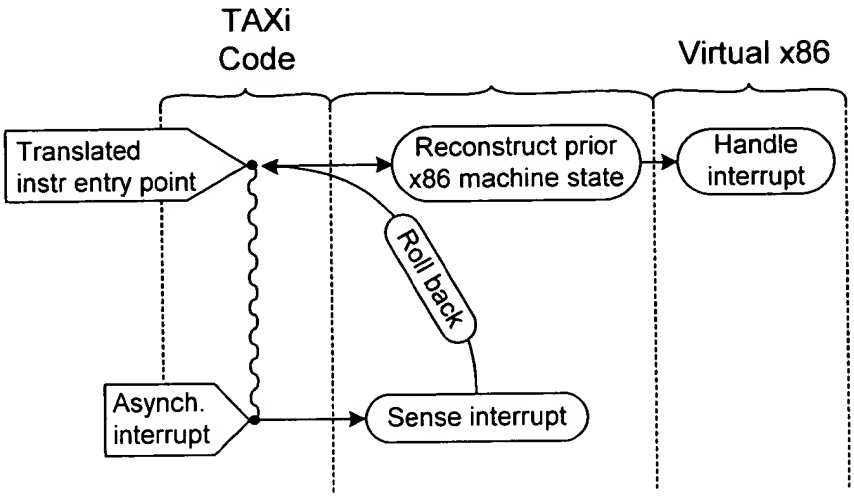
This section of Yate's disclosure refers to "case 353." This is explained in part of *Yates'* Figure 3J (emphasis added):

```
} else if EPC.Taxi_Active {
    // A Taxi translated version of some X86 code was running.  Taxi will rollback to an
    // x86 instruction boundary.  Then, if the rollback was induced by an asynchronous external
    // interrupt, Taxi will deliver the appropriate x86 interrupt...
```

353

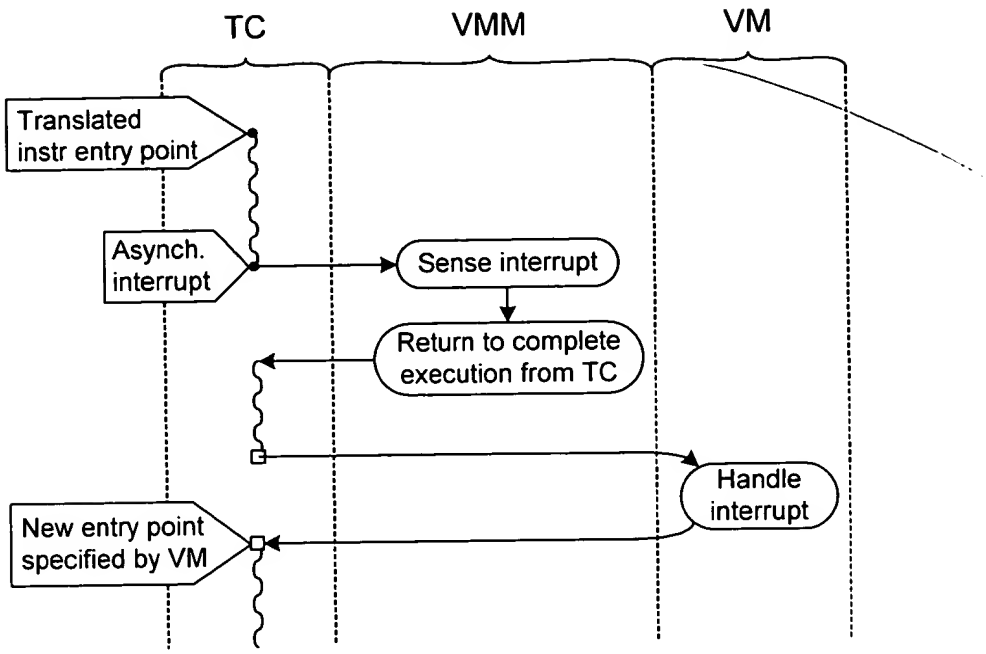
Thus, when *Yates'* system is executing its version of translated code and an asynchronous exception arises, execution is **rolled back** to the entry point for the current translated instruction sequence, and no execution of anything relating to the original instruction stream is resumed until **after** the interrupt is delivered. Note, moreover, that when execution does continue, it appears to occur in the **hardware** converter 136 and not in the Tapestry system's previous context of binary translation.

The following diagram illustrates *Yates'* system's procedure, in which the vertical "wavy" line indicates translated (TAXi) instructions being executed:




Note that no purpose would be served by reconstructing and supplying an earlier x86 machine state *unless* execution is rolled back: If execution of binary-translated code were to proceed, the previous machine state would be irrelevant.

In contrast, according to the applicant's invention, when an asynchronous exception is sensed, execution of the current translated sequence is resumed – **in binary translation** – and is allowed to continue to completion, **before** the exception is handled. The following diagram illustrates the applicant's method, in which "TC" indicates "Translation Cache," and, as before, the wavy vertical line segments indicate execution of cached, binary-translated instructions:




Claims 1 and 16 have now been amended to explicitly state that: if the sensed exception is asynchronous, resuming to completion the execution of the target instruction sequence in binary translation before handling the asynchronous exception. Claim 15 already included this limitation in similar words (see claim 15, lines 39-42 as numbered above). All of the independent claims therefore make clear a feature that is not only lacking in *Yates*, but that *Yates* specifically teaches away from.

In claim 16, mention of the *resume* software component (Figure 3, component 342) has been added as being the component of the intermediate software layer that sees to it that execution of the target instruction sequence in binary translation is resumed to completion before handling the asynchronous exception. Recitation of the intermediate software layer (for example, the VMM) has also been added since that is the software layer the *resume* component preferably is part of.

"Single Common Strategy" Obvious But Not the Invention

The Examiner also wrote:

> "It would have been obvious to one of ordinary skill in the art at the time of invention was made to improve upon exception handler taught by *Yates*' provides a flexibility to have a single common strategy for processing all exceptions raised during the binary translation either by source or target ISA and execute exception handler on either source ISA or target ISA."

Only one single, common strategy would be possible in conventional processor architectures, namely, immediate handling of both synchronous and asynchronous exceptions without resuming and continuing execution of a given instruction stream. This is so because of the very definition of a synchronous exception.

Consider the definition of a synchronous exception as given in the applicant's specification, on page 3, lines 14-17:

> Synchronous exceptions are exceptions generated as a direct result of the attempt to execute the (target) instruction. Common examples of synchronous exceptions include page faults, general exception faults, division-by-zero errors, and illegal instruction faults.

Assume that execution of a sequence of translated instructions included an illegal instruction, or caused a general exception fault, or attempted division by zero. If handling of such a synchronous exception were delayed and execution of the sequence of translated instructions were resumed to completion, then the results on the state of the machine would be different from the specified behavior of the processor, leading to unpredictable behavior at best and catastrophic results at worst.

Because of the nature of synchronous exceptions, modern hardware architectures (for example, x86) therefore require that all synchronous exceptions must be handled without delay. In fact, this requirement is part of the specification of the architecture and thus part of the accepted definition of synchronous exceptions.

In contrast, as stated in the specification, page 3, lines 18-24:

> Asynchronous exceptions (also called interrupts) are, in contrast, generated by the processor as the result of an external event. Examples of such asynchronous exceptions include device completion interrupts, timed interrupts, disk interrupts, and inter-processor interrupts requested by another processor. Asynchronous exceptions can thus also be considered to be a form of "external interrupts" since they are typically caused by some external device and are typically not "errors," inasmuch as they signal or correspond to some desired device action.

The "obvious" way to handle even asynchronous exceptions is to do what *Yates* does, that is, not to complete execution of a current translated sequence but rather to "start over" *after* handling the asynchronous exception. The applicant agrees that it would be obvious to handle both synchronous and asynchronous exceptions in this manner – *Yates* illustrates that this method is known.

The applicant has found a different solution, however, one that does not require rolling back execution of translated instruction sequences when asynchronous exceptions occur during their execution. The primary novel aspect of the applicant's invention is that it does *not* require a single common strategy for handling the different types of exceptions. Consequently, in the context of binary translation, only the applicant's invention provides the advantage of continued *forward* execution progress for translated code when handling asynchronous exceptions that arise during the execution of the sequence.

**Finality of Rejection Should Be Withdrawn**

The remarks presented above are clarifications of what was explained in the response to the previous Office action. The changes to the claims also simply clarify already claimed features, in order to place the claims in condition either for allowance or appeal. Consequently, the applicant respectfully requests the Examiner to enter this amendment and allow the claims without requiring the applicant to file a Request for Continuing Examination.

**Conclusion**

The applicant's invention as defined in the independent claims includes at least one feature that is not found or suggested in the cited prior art, and that provides a substantial technical advantage. As such the applicant respectfully submits that the independent claims should be allowed, as well as the remaining, dependent claims that simply further narrow the definition of the invention.

**Applicant No Longer a Small Entity**

The applicant is no longer entitled to claim small entity status for purposes of paying reduced fees. If, as the applicant respectfully submits is justified, the next action by the Office is a Notice of Allowance, then the Office should take this into account when indicating the issue fee due.

**Change of Attorney Name**

Along with the 20 October 2003 response to the previous Office action, the applicant's attorney included a letter explaining that the attorney had legally changed his last name from "Slusher" to "Pearce." This most recent Office action was still mailed to Jeffrey "Slusher," however. Please make note of this change. Thank you.

Date: 2 February 2004

Respectfully submitted,

Jeffrey Pearce

34825 Sultan-Startup Rd.
Sultan, WA 98294
Phone & fax: (360) 793-6687

Jeffrey Pearce
Reg. No. 34,729
Attorney for the Applicant